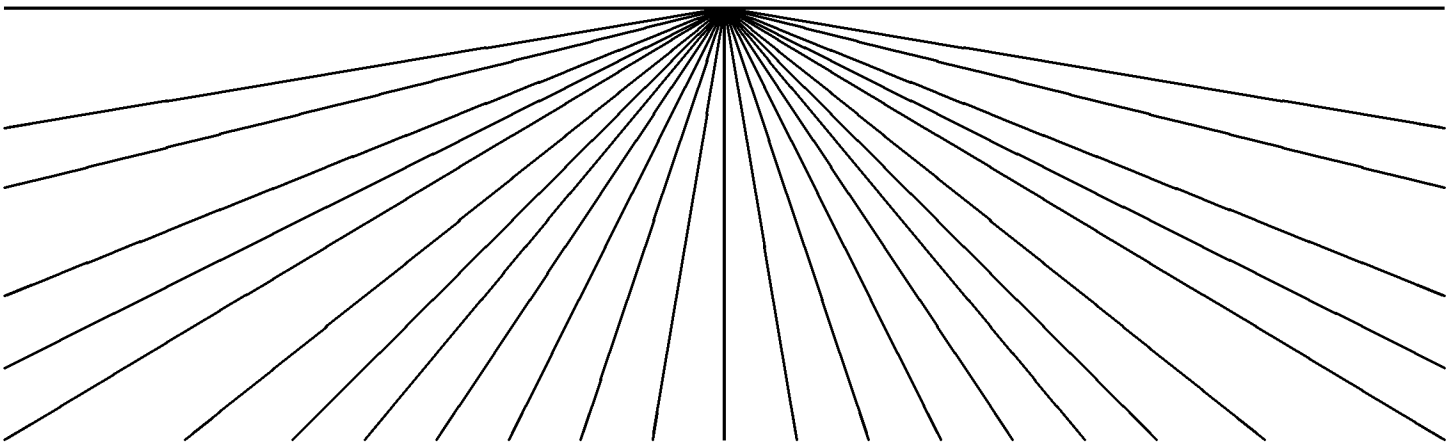
facultad de informática

universidad politécnica de madrid

The AND-Prolog Compiler System — Automatic Parallelization Tools for LP

F. Bueno
D. Cabeza
M. García de la Banda
M. Hermenegildo

TR Number DIA/CLIP5/93.0



The AND-Prolog Compiler System — Automatic Parallelization Tools for LP

Technical Report Number: DIA/CLIP5/93.0

June 1993

Authors

F. Bueno, D. Cabeza, M. García de la Banda and M. Hermenegildo

CLIP is the Computational Logic, Implementation and Parallelism Group.
Universidad Politécnica de Madrid (UPM), Facultad de Informática,
28660–Boadilla del Monte, Madrid — Spain.
clip@dia.fi.upm.es

Keywords

Compilation techniques, Abstract Interpretation, Program Transformation, Parallelism

Acknowledgements

This work was funded in part by ESPRIT project #6707 "PARFORCE" and by CICYT project TIC93-0976-CE.

Abstract

This report presents an overview of the current work performed by us in the context of the efficient parallel implementation of traditional logic programming systems. The work is based on the &-Prolog System, a system for the automatic parallelization and execution of logic programming languages within the Independent And-parallelism model, and the global analysis and parallelization tools which have been developed for this system. In order to make the report self–contained, we first describe the "classical" tools of the &-Prolog system. We then explain in detail the work performed in improving and generalizing the global analysis and parallelization tools. Also, we describe the objectives which will drive our future work in this area.

Contents

1 Introduction

Efficient, practical, high–performance multiprocessors are now a market reality. However, the amount of software that can exploit the performance potential of these machines is still very small. This is largely due to the difficulty in mapping the inherent parallelism in problems onto different multiprocessor organizations. In general, there are at least two ways in which such a mapping can be performed: it can be done explicitly in the program by the user, or it can be automatically uncovered by a "parallelizing" compiler.

The latter approach seems to be desirable, since it avoids burdening the programmer with low–level, machine–dependent details. However, the capabilities of current parallelizing compilers are relatively limited, specially in the context of conventional programming languages. The former can be used when the programmer has a clear understanding of how the parallelism in the problem can be exploited in the target architecture. However, the task of correctly determining the *data dependencies* among those parts and the sequencing and synchronization needed to reflect such dependencies is proving to be very difficult and error–prone. This was also pointed out by Karp [28] who states that "the problem with manual parallelization is that much of the work needed is too hard for people to do. For instance, only compilers can be trusted to do the dependency analysis needed to parallelize programs on shared–memory systems."

Therefore, the best programming environment would appear to be one in which the programmer can freely choose between only concentrating on the conventional programming task itself (letting a parallelizing compiler uncover the parallelism in the resulting program) or, alternatively, performing, in addition, the task of explicitly annotating parts of the program for parallel execution. In the latter case, the compiler should be able to aid the user in the dependency analysis and related tasks. Ideally, different choices should be allowed for different parts of the program.

Declarative languages and, in particular, logic programming languages, require almost no explicitation of control (thus making easier the mapping between the statement of the problem and its coding). In addition, their semantics makes them appropriate for compile–time analysis and program parallelization. In other words, such programs preserve more the intrinsic parallelism in the problem, make it easier to extract in an automatic fashion, and allow the techniques being used to be proved correct.

It is our thesis that through advanced compiler techniques, such as abstract interpretation, automation of parallelization is indeed feasible for languages that have a declarative foundation. Furthermore, we believe that the development of these techniques will give in addition further insight in our understanding of how to parallelize other programming paradigms.

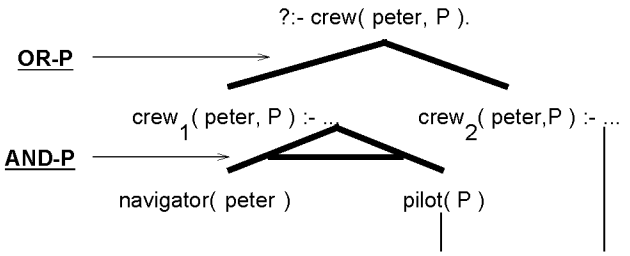


Figure 1: Or– and And–parallelism in Logic Programs

The aim of this work is to present an overview of the current work performed by us in the context of the efficient parallel implementation of traditional logic programming systems. The work is based on the &-Prolog System, a system for the automatic parallelization and execution of logic programming languages within the Independent And-parallelism model, and the global analysis and parallelization tools which have been developed for this system.

In doing this and in order to be self-contained we will start by introducing the subject, determining the issues which form the core of our research effort. Then we will briefly present the &-Prolog System and the tools which were already developed before the beginning of the present project. Later we will explain in detail the work performed in improving and generalizing the global analysis and parallelization tools. Finally, we will describe how the integration of all these improved tools is being performed in a modular framework and the objectives which will drive our future work in this area.

It is important to note that the work presented here will be focussed on the techniques developed for improving and generalizing the partitioning tools defined for *traditional* logic programming languages. Thus, issues of support for constraints and concurrency will be not addressed here since they are the the core of the work presented in *ParForce* deliverable D.WP1.2.1.M1.5.

2 Logic Programming and And–Parallelism

The two main types of parallelism which can be exploited in logic programs are well known[8]: (1) and–parallelism and (2) or–parallelism. Several models have been proposed to take advantage of such opportunities (see, for example, [16], [38], [2], [25], [29], [49], [20], [47], [43], [1] and their references). The following example and Figure 1 serves to illustrate where parallelism is available:

```
crew(X,Y) :- navigator(X), pilot(Y).
crew(X,Y) :- mechanic(X), pilot(Y).
```

As can be seen in Figure 1 there are two *alternative* ways to try to satisfy the goal
`?:-crew(peter,P).`
corresponding to the two clauses in the definition of `crew/2`. It is possible to have different processors proceed simultaneously with such alternatives. The resulting parallelism is called or–parallelism. Thus, or–parallelism corresponds to the parallel exploration of branches in the proof tree corresponding to different clauses which match a given goal. Consider now the execution of one of the alternatives, for example $\texttt{crew}_1$ in Figure 1. Now, to satisfy $\texttt{crew}_1$ *both* `navigator(peter)` and `pilot(P)` need to be satisfied. Parallelism can also be achieved if these two goals are executed in parallel. This is called and–parallelism, i.e., and–parallelism refers to the parallel execution of the goals in the body of a clause.[1]

Significant research effort has been and is being applied to developing or–parallel execution models (see, for example, [46] and its references). The associated performance studies have shown good performance for non–deterministic programs in a number of practical implementations [43, 1]. The resulting speedups obtained over state–of–the–art sequential systems in non–deterministic programs support the thesis defended in this paper for this class of programs. The &–Prolog approach is complementary to that of or–parallelism: start with the exploitation of (independent) and–parallelism, develop compilation (parallelization) and implementation technology, and then extend the system to exploit or– and (dependent) and–parallelism. We choose to study (independent) and–parallelism first because of its ability to exploit parallelism in deterministic programs (informally, those which only make trivial use of backtracking) as well as non–deterministic programs, its compatibility with full Prolog functionality [25, 21], its inherent efficiency in resource management, and the relative lack of implementation technology for this type of parallelism. Together, or–parallelism and and–parallelism appear to be capable of producing speedups in a large class of programs [40].

2.1 Independent And–Parallelism and &–Prolog

As mentioned before, and–parallelism refers to the concurrent execution of goals in the bodies of clauses. Let's consider the following program

```
crew(X,Y) :- navigator(X) & pilot(Y).
crew(X,Y) :- mechanic(X) & pilot(Y).
```

where "&" denotes and–parallel execution. We refer to the language resulting from incorporating the "&" to Prolog as "&–Prolog" (see Section 2.3). Assume now that the query is
`?:- crew(C1,C2).`
According to the semantics of "&" `navigator(C1)` and `pilot(C2)` would be executed in parallel. Note that this presents no problems since the tasks of finding a candidate navigator and a candidate pilot are independent. However, problems arise in cases

[1] Or, more formally, to the concurrent search for proofs of goals in the current resolvent, see [21].

where such independence does not hold. Let's assume that the `pilot/1` clause is defined as "`pilot(P):- license(P), medical(P).`" and parallelized as "`pilot(P):- license(P) & medical(P).`" Consider the execution of the body of this clause. Now `license(C2)` and `medical(C2)` are "dependent," i.e. they share an unbound variable, C2, and should generate unifiable bindings for it. There are two basic alternatives at this point: generate all solutions for `license(C2)` and `medical(C2)` and perform a join (intersection) operation, or sequence them so that one is the producer of `C2` and the other the consumer, as in Prolog (this is also referred to as the generator–consumer or "nested loops" approach).

The first solution is generally perceived as impractical due to the need for excessive storage and the generation of much additional work. Independent And–Parallelism (IAP) [21, 23, 16, 25] selects the second method above, i.e. given two goals `p` and `q` in the body of a clause, "`p & q`" is a correct annotation iff `p` and `q` are mutually independent. Independent goals are defined as goals which do not share any variables *at run–time*. In other words, goals which are dependent, like `license(C2)` and `medical(C2)` above since they share the variable `C2`, should be sequenced (using ","). It should be noted that, as an alternative to IAP, pipelining values between dependent goals (dependent and–parallelism) offers the possibility of some additional parallelism in some cases but at the price of increased complexity —and overhead— in the implementation. Additionally, it is generally accepted that it only makes sense to run *determinate* dependent goals in parallel [3] while IAP allows parallel execution of non–determinate goals. Therefore, and as mentioned before, our approach is to first prove feasibility for the case of IAP and later extend the capabilities of the system to support or–parallelism and dependent and–parallelism.

Given the description of IAP above, we could conclude that running `license(C2)` and `medical(C2)` in parallel in the previous example was incorrect because they were dependent and that a correct annotation for the `crew/pilot/...` example would be

```
crew(X,Y) :- navigator(X) & pilot(Y).
pilot(P)  :- license(P), medical(P).
```

In fact, this annotation is appropriate only for the particular query considered. Consider the query `crew(N,peter)`. Now the goals in the body of the `pilot/1` clause are `license(peter), medical(peter)` which are independent (they share no variables) and could have been run in parallel. But suppose that the query in the above example is changed to `crew(Loner,Loner)`, i.e., it is desired to have a one–person crew. Now the goals `navigator(Loner)` and `pilot(Loner)` are dependent and the `&` in `crew/2` is incorrect for this query!

The conclusion is that goal independence is a function of the run–time instantiations of the variables in the goals being considered, and, therefore, is in general query–dependent. Efficient annotation requires either a priori knowledge of the binding patterns of the variables in the programs at run–time, or checks which can dynamically

determine at run–time which goals should be executed in parallel. As will be shown, the latter can be done quite simply within the "&–Prolog" language.

2.2 Strict and Non–Strict Independent And–Parallelism

As discussed above, Independent And–Parallelism follows the producer–consumer approach in implementing and–parallelism. Goals in the body of a clause are run in parallel iff they are *independent* at run–time. Goals which are not independent at run–time are called *dependent* and are executed sequentially in left–to–right order,[2] since we want to preserve Prolog semantics and efficiency.

In the broadest interpretation of IAP, goals are deemed to be independent iff they cannot affect each other's search space.[3] In traditional, i.e. *strict* IAP [21], this translates to the requirement that *run–time instantiations of these goals do not share any variables*. Define *vars(g)* to be the set of all variables in g. Two goals g_1 and g_2 are thus defined to be strictly independent if $vars(g_1) \cap vars(g_2) = \emptyset$.

On the other hand, one can relax this requirement by defining *non–strict independence* [23] as follows. Two goals are non–strictly independent iff

- if they are strictly independent, or
- if their run–time instantiations do share variables, but these goals do not "compete" for the bindings of these variables.

Within this context, two goals compete for a variable if they try to bind it, even if this occurs in a failing branch and the binding is never seen after the goals' success.

2.3 The &–Prolog Language

The &–Prolog language is a vehicle for expressing and implementing strict and non–strict Independent And–Parallelism. &–Prolog is essentially Prolog, with the addition of the parallel conjunction operator "`&`" (used in place of "," (comma) when goals are candidates for safe parallel execution) and a set of parallelism–related builtins, which includes several types of groundness and independence checks, and synchronization primitives. Combining these primitives with the normal Prolog constructs, such as "`->`" (if–then–else), users can conditionally trigger parallel execution of goals. &–Prolog is capable of expressing both restricted [16] and unrestricted IAP (through the use of the `wait` primitives [32]). For syntactic convenience, an additional construct is also provided: the *Conditional Graph Expression (CGE)*. A CGE has the general

[2] i.e. the leftmost goal is the producer and the other goals are the consumers.

[3] This independence ensures that a particular implementation of IAP can execute them in parallel while being able to guarantee an important "no–slowdown" property w.r.t. the sequential execution [23] — namely that the time required for the parallel execution is less or equal to the time for the sequential execution.

form (*i_cond* `=>` $goal_1$ `&` $goal_2$ `&` ... `&` $goal_N$) where the $goal_i$ are either normal Prolog goals or other CGEs and *i_cond* is a condition which, if satisfied, guarantees the mutual independence of the $goal_i$s. The operational meaning of the CGE is "check *i_cond*; if it succeeds, execute the $goal_i$ in parallel, otherwise execute them sequentially." &–Prolog if–then–else expressions and CGEs can be nested in order to create richer execution graphs. *i_cond* can in principle be any &–Prolog goal but is in general either `true` ("unconditional" parallelism) or a conjunction of checks on the groundness or independence of variables appearing in the $goal_i$s. An &–Prolog annotation for the example in the previous section is, for instance

```
crew(X,Y) :- (indep(X,Y) -> navigator(X) & pilot(Y)
                         ;  navigator(X),  pilot(Y) ).
pilot(P)  :- (ground(P)  -> license(P) & medical(P)
                         ;  license(P),  medical(P) ).
```

For example, in the first clause the variables `X` and `Y` are checked at run–time. If the terms they are bound to are independent (have no variables in common) they will execute in parallel, an sequentially otherwise.

2.4 The &–Prolog System

Figure 2 shows the conceptual structure of the &–Prolog system. It is a complete Prolog implementation, offering full compatibility with the DECsystem–20/Quintus Prolog ("Edinburgh") standard, plus supporting the &–Prolog extensions. The user interface is the familiar one with an on–line interpreter and compiler. At the system prompt, and following the objective of supporting both automatic parallelism and user expressed parallelism, the user can choose to input (consult/compile) "conventional" Prolog code. In this mode users are unaware (except for the difference in performance) that they are using anything but a conventional Prolog system. A compiler switch determines whether or not such code will be parallelized. Alternatively the user can provide Prolog code which is annotated with &–Prolog constructs. This can be done for a whole file, a procedure, or a single clause, while the rest of the program can still be parallelized automatically. The compiler still checks the user supplied annotations for correctness, and provides the results of global analysis to aid in the dependency analysis task.

In the compiler, input code is analyzed by four different modules as follows:

- The **Annotator**, or "parallelizer," performs local *dependency* analysis on the input code. In addition, it can also have information about the possible run–time substitutions ("variable bindings") at all parts in the program from the global analyzer and whether or not a predicate has side–effects from the side–effect analyzer. It uses all this information to annotate the input code for parallel execution. Its output is an annotated &–Prolog program.

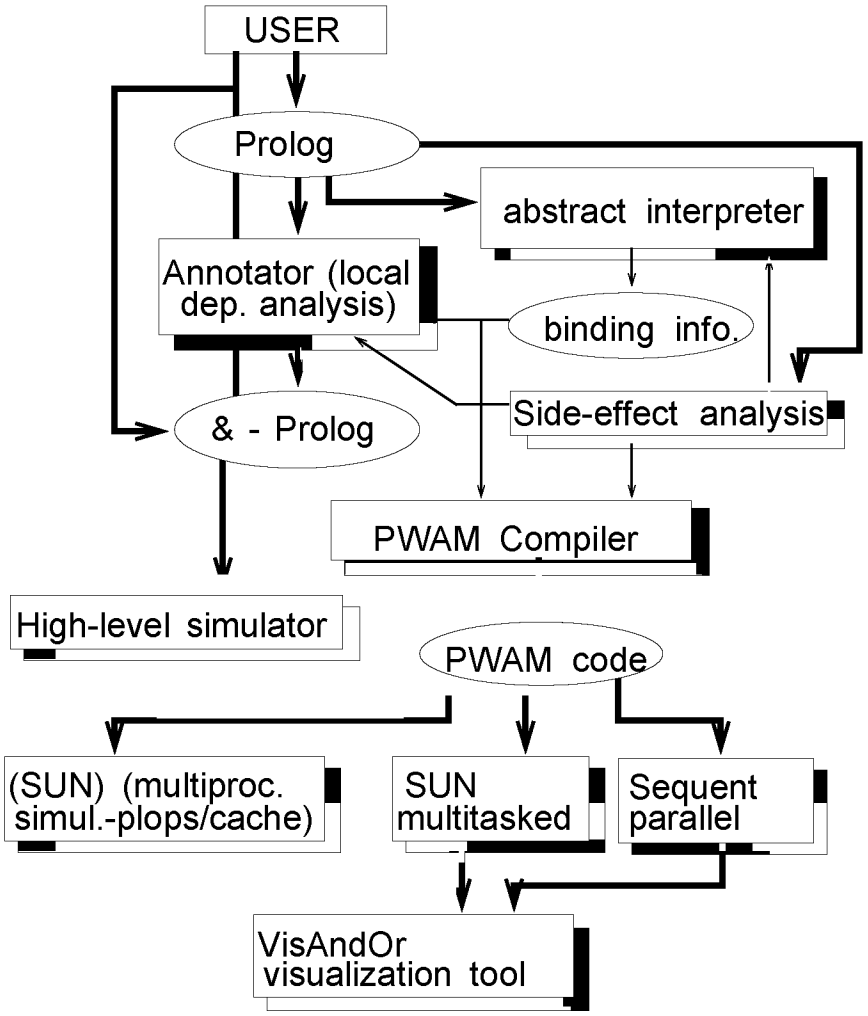


Figure 2: &-Prolog System Architecture and Performance Analysis Tools

- The **Global Analyzer** interprets the given program over an abstract domain (specifically designed to precisely highlight dependence information) and infers information about the possible run–time substitutions at all points of the program. This information, which is obviously not obtainable from local clause–at–a–time analysis alone, is used by the annotator, as mentioned before, which would then generate a potentially more efficient annotation.

- The **Side–effect Analyzer** annotates each non–builtin predicate and clause of the given program as *pure*, or as containing or calling a *side–effect*. This information is used by the annotator to introduce semaphores into the clauses, if

necessary, in order to correctly sequence such side–effects. The techniques used for sequencing side–effects at the &–Prolog level and at the abstract machine level are presented in [32].

The &–Prolog code (annotated Prolog) produced can be saved for analysis by a high–level simulator [40] which determines the available parallelism and expected speedup given the annotations and a set of assumptions about the cost of operations on the target parallel machine. Alternatively, rather than being used as input for the high–level simulator, the &–Prolog code (annotated Prolog) is translated into low–level PWAM (Parallel Warren Abstract Machine) byte code by the PWAM compiler for actual execution on the PWAM abstract machine. The PWAM machine is an extension of the Warren Abstract Machine (WAM) architecture [45] capable of executing logic programs in parallel as determined by &–Prolog's annotations.

Having defined &–Prolog, and shown its suitability for *expressing* independent and–parallel execution, some issues still remain to be dealt with. In particular, how correct and efficient annotations can be generated automatically [34], how the peculiar characteristics of practical Prolog programs (for example, those with side–effects) are dealt with [32] and how much parallelism can be obtained from such automatic annotations. This will be the issues discussed in the next two sections.

3 Global Analysis Tools

The *independence* and *groundness* checks used in conditional expressions in &–Prolog can take up a considerable amount of execution time and thus form a substantial overhead to the parallel execution of a program. Thus, the effort should be aimed at eliminating as many checks as possible by gathering highly accurate information at compile–time about the groundness and independence of the terms to which program variables will be bound at run–time. This can be achieved through *global* analysis of the given program at compile–time using the technique of *abstract interpretation.*

The technique of abstract interpretation for flow analysis of programs in imperative languages was first presented in a sound mathematical setting by Cousot and Cousot [10] in their landmark paper. Later, it was shown by Bruynooghe [4], Jones and Sondergaard [27], and Mellish [31] that this technique can be extended to flow analysis of programs in logic programming languages. In this framework a program analysis is viewed as a non–standard, abstract semantics defined over a domain of data descriptions. An abstract semantics is constructed by replacing operations in a suitable concrete semantics with corresponding abstract operations defined on data descriptions. Program analyses are defined by providing finitely computable abstract interpretations which preserve interesting aspects of program behaviour.

In the case of logic programming languages, "data" corresponds to substitutions and atoms. The basic operations on data typically include unification, composition

of substitutions and projection of substitutions onto variables of interest. Proving the safety of an abstract unification function is the major step in proving the safety of abstractions for logic programs.

Specific algorithms for such global analysis in logic programs have been given by a number of researchers ([15], [30], [39], [44], [48], ...). With few exceptions, these schemes are geared towards optimizing the *sequential* execution of logic programs. They focus on computing information about the arguments of *predicates* used in the program, such as the *mode* and the *type* of an argument. Also, although variable sharing is dealt with in these methods as needed in order to preserve the correctness of the approach, it is not generally regarded as one of the main outputs of the analysis and often computed in a very conservative way [14]. However, as shown above, variable sharing information can be of the utmost importance for a compiler which targets execution in a system which supports Independent And–Parallelism (IAP).

Furthermore, such compiler requires information for all points in the program, rather than globally for each procedure. Note that this requirement is actually the only significant departure from conventional abstract interpretation involved in supporting IAP: as shown in [22], in IAP, the execution of a program in parallel produces the same answer substitutions as the standard sequential execution model. For this reason the global analyzers developed for the &-Prolog systems have been integrated in an abstract interpretation framework capable to provide such information.

The main subject of this section will be to describe such framework and the global analyzers developed for the &-Prolog System. In doing this, we will start by briefly describing the standard framework of abstract interpretation as defined in [10] in terms of *Galois insertions*. Then we will describe in a bit more detail the particular abstract framework and fixpoint algorithm in which the analyzers developed for the &–Prolog system are implemented. Finally we will present those analyzers pointing out the impact that the information obtained by each of those analyzers have on the accurate determination of independence.

3.1 Abstract Interpretation

Definition 1 *Galois insertion*
A Galois insertion is a quadruple (E, α, D, γ) *where:*

1. $(E, \sqsubseteq_E)$ *and* $(D, \sqsubseteq_D)$ *are complete lattices called* concrete *and* abstract domains *respectively;*
2. $\alpha : E \rightarrow D$ *and* $\gamma : D \rightarrow E$ *are monotonic functions called* abstraction *and* concretization functions *respectively; and*
3. $\alpha(\gamma(d)) = d$ *and* $e \sqsubseteq_E \gamma(\alpha(e))$ *for every* $d \in D$ *and* $e \in E$.

In general only one of $\left\{ \alpha, \gamma \right\}$ need be specified since in principle a "best possible"

α can always be determined for a given γ and vice versa.

The following specifies the notion of approximation which is then lifted from the primitive domains to function domains:

Definition 2 *approximation*
Let (E, α, D, γ) be a Galois insertion and let $\mu : E \to E$ and $\mu^{\mathcal{A}} : D \to D$ be monotonic functions. We say that $d \in D$ γ-approximates *$e \in E$, denoted $d \propto_\gamma e$, if $e \sqsubseteq_E \gamma(d)$*[4]*. We say that $\mu^{\mathcal{A}}$* γ-approximates *μ, denoted $\mu^{\mathcal{A}} \propto_\gamma \mu$, if $\forall d \in D.\ e \in E.\ d \propto_\gamma e \Rightarrow \mu^{\mathcal{A}}(d) \propto_\gamma \mu(e)$.*

Concrete semantics are typically defined as least fixed points of an operator on programs. Typically, the meaning of a program P may be expressed as $[\![P]\!] = lfp(f_P)$ where $f_P : Den \to Den$ is a monotonic operator on a domain of denotations Den. A program analysis will typically be defined by introducing an appropriate Galois insertion $(Den, \alpha, Den^{\mathcal{A}}, \gamma)$ and constructing an approximation $f_P^{\mathcal{A}} : Den^{\mathcal{A}} \to Den^{\mathcal{A}}$ of f_P so that the least fixed point of $f_P^{\mathcal{A}}$ is finitely computable. This construction often takes a systematic approach which involves replacing the basic operations in the concrete semantic operator f_P by corresponding abstract operations in $f_P^{\mathcal{A}}$ (e.g., [12, 37]). Given that these abstract operations approximate the concrete operations it is generally straightforward to prove that the derived abstract semantic operator approximates the concrete semantic operator. The fundamental theorem of abstract interpretation provides the following result:

Theorem 1
Let (E, α, D, γ) be a Galois insertion and let $\mu : E \to E$ and $\mu^{\mathcal{A}} : D \to D$ be monotonic functions such that $\mu^{\mathcal{A}}$ γ-approximates *μ. Then $lfp(\mu^{\mathcal{A}}) \propto_\gamma lfp(\mu)$.*

The *"art"* of abstract interpretation can be described as involving the following steps: (1) to choose an appropriate concrete semantics; (2) to identify a suitable notion of data–description; and (3) to provide good approximations of the basic operations in the concrete semantics. Once this is done the foundation is laid for deriving, more or less automatically, a semantics based program analysis. Applying suitable optimizations to the fixpoint algorithm used in the description domain, an analysis that is also efficient can be built essentially automatically from it [4, 36]. In the case of logic programs the main step is to provide a notion of abstract substitutions and an abstract unification algorithm. Other operations include "projection" and "composition" which safely project (i.e., on a finite set of variables) and compose descriptions.

3.2 Framework

The input to the abstract interpreter is a set of clauses (the program) and set of "query forms." As mentioned, one of the main requirements for the abstract interpreters

[4] Or alternatively if $\alpha(e) \sqsubseteq_D d$.

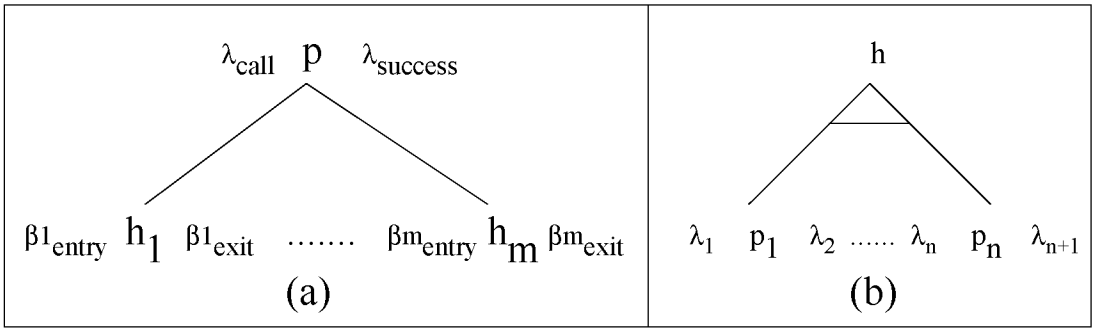


Figure 3: Illustration of the abstract interpretation process

developed for the &-Prolog System is to compute the abstract information at all points of all the clauses that would be used while answering all possible queries which are concretizations of the given query forms. Thus, it is convenient to give different names to abstract substitutions depending on the point in a clause to which they correspond. Consider, for example, the clause h :- $p_1, \ldots, p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal $p_i, 1 \leq i \leq n$ in this clause. See figure 3(b).

Definition 3 *λ_i and λ_{i+1} are, respectively, the* abstract call substitution *and the* abstract success substitution *for the subgoal p_i. For this same clause, λ_1 is the* abstract entry substitution *(also represented as β_{entry}) and λ_{n+1} is the* abstract exit substitution *(also represented as β_{exit}).*

Control of the interpretation process can itself proceed in several ways, a particularly useful and efficient one, able at providing information at all program points, being to essentially follow a top–down strategy starting from the query forms. Several frameworks for doing abstract interpretation in logic programs follow along these lines. One such framework is described in detail for example in [4]. In a similar way to the concrete top–down execution, the abstract interpretation process can then be represented as an abstract AND–OR tree, in which AND–nodes and OR–nodes alternate. A clause head h is an AND–node whose children are the literals in its body $p_1, \ldots, p_n$ (figure 3(b)). Similarly, if one of these literals p can be unified with clauses whose heads are $h_1, \ldots, h_m$, p is an OR–node whose children are the AND–nodes $h_1, \ldots, h_m$ (figure 3(a)). In building the abstract AND–OR tree for a given program and a goal, the abstract interpreter has to repeatedly execute the *basic* step of computing the *success* substitution of a subgoal whose *call* substitution is given. Given a subgoal p, its call substitution λ_{call} and clauses $C_1, \ldots, C_m$ whose heads unify with p, a naïve approach to executing this basic step would be to build the subtree for p in a top–down fashion:

- Project λ_{call} on to the variables in p to obtain λ, the projected call substitution for p.

- For each clause C_i:
  - compute its entry substitution,
  - compute its exit substitution by recursively computing the success substitutions for each of its subgoals in a left–to–right fashion,
  - compute λ'_i, the projected success substitution for p from clause C_i,
- Compute λ', the projected success substitution for p by taking the *least upper bound* (LUB) of $\lambda'_i, 1 \leq i \leq m$
- Extend λ' to $\lambda_{success}$, the success substitution for p.

The overall abstract interpretation scheme described works in a relatively straightforward way if the program has no recursion. Consider, on the other hand, a recursive predicate p. If there are two OR–nodes for p in the abstract AND–OR tree such that they are identical (i.e., they have the same atoms), one is an ancestor of the other, and the call substitutions are the same for both, then the abstract AND–OR tree is *infinite* and an abstract interpreter using the simple control strategy described above will not terminate.

The goal of the fixpoint algorithm is then to facilitate the computation of the abstract information in such cases without going into an infinite loop. The basic idea behind such algorithm is as follows:

- Compute the *approximate* value of λ' using the non–recursive clauses $C_1, \ldots, C_r$ for p and record this value in a *memo table* [18].
- Construct the subtree for p, using the approximate value of λ' from the memo table, if necessary.
- Update the value of λ' using p's subtree. Update p's subtree to reflect this change and compute the new value of λ' again. Repeat this step until the value of λ' doesn't change, i.e., it has reached *fixpoint*.

The aim of the memo table is to store possibly incomplete results obtained from an earlier round of iteration. The memo table has an entry for each *node*, i.e for each subgoal with a distinct atom and a distinct (projected) call substitution (modulo renaming of the variables) that occurs in the abstract AND–OR tree. In addition, each entry contains the projection of its success substitution on the subgoal variables (λ'), characterization of this information indicating if the information is *complete*, *approximate* or *fixpoint* (the meaning of these labels will be explained below), and a unique ID identifying the node in the abstract AND–OR tree.

The global analysis tools developed for the &-Prolog System use a highly optimized fixpoint computation algorithm based on the scheme above, which is described in [36].

3.3 Sharing analyzer

The Sharinganalyzer defined in [36] was the first analyzer aimed at inferring accurate sharing information as the previous step for automatically parallelization. The approach to defining abstract substitutions was entirely different to that followed by the traditional analyzers. It was not *per se* interested in the set of terms that a program variable is bound to at a point in a clause. Rather, it was interested in the *sharing* of variables among the sets of terms that program variables are bound to. For example, let X and Y be the program variables in a clause. The abstract substitution in the Sharingabstract interpreter should tell us whether the sets of terms that X and Y are bound to, share any variables or not. As discussed above, this information will help to eliminate groundness/independence checks.

The abstract substitution for a clause are then defined as *a set of sets of program variables* in that clause. This follows an approach initially suggested in [26]. For the example clause of the previous paragraph, the value of an abstract substitution may be $\{\{X\},\{X, Y\}\}$. This abstract substitution corresponds to a set of substitutions in which X and Y are bound to terms T_X and T_Y such that (1) a variable occurs in both T_X and T_Y (this corresponds to the element $\{X, Y\}$) and (2) a variable occurs only in T_X (this corresponds to the element $\{X\}$).

Below, we formally define the abstract substitution $\mathcal{A}(\theta)$ which corresponds to a concrete substitution θ. The basic idea behind this definition is as follows: a set S of program variables appears in $\mathcal{A}(\theta)$ iff there is a variable Z which occurs in each member of S under θ. Thus, a program variable is ground if it does not appear in any set $\mathcal{A}(\theta)$, and two program variables are independent if they do not appear together in any set in $\mathcal{A}(\theta)$. In other words, each set in the abstract substitution containing variables $v_1, \ldots, v_n$ represents the fact that there may be one or more shared variables occurring in the terms to which $v_1, \ldots, v_n$ are bound. If a variable v does not occur in any set, then there is no variable that may occur in the terms to which v is bound and thus those terms are definitely ground. If a variable v appears only in a singleton set, then the terms to which it is bound may contain only variables which do not appear in any other term.

Before we formally define the abstraction function $\mathcal{A}$, let us first review some basic definitions about substitutions. A substitution for a clause is a mapping from the set of program variables in that clause (*Pvar*) to terms that can be formed from the universe of all variables (*Var*), the constants and functors in the given program. The domain of a substitution θ is written as $dom(\theta)$. We consider only idempotent substitutions. The instantiation of a term `T` under a substitution θ is denoted as `T`θ. `var(T`θ`)` denotes the set of variables in Tθ. *Subst* is the set of all substitutions which map variables in *Pvar* to terms comprising of variables in *Var*, constants and functors in the given program. *Asubst* is the set of all abstract substitutions for a clause. $Asubst = \wp(\wp(Pvar))$ where $\wp(S)$ denotes the powerset of S. The function *Occ* takes two arguments, θ (a substitution) and U (a variable in *Var*) and produces the set of all program variables

$V \in Pvar$ such that U occurs in $var(V\theta)$ i.e

$$Occ(\theta, U) = \{V \mid V \in dom(\theta) \wedge U \in var(V\theta)\}$$

Definition 4 (Abstraction of a substitution)

$$\mathcal{A} : Subst \rightarrow Asubst$$

$$\mathcal{A}(\theta) = \{Occ(\theta, U) \mid U \in Var\}$$

Example: Let $\theta = \{W/a, X/f(A,B), Y/g(B), Z/C\}$. $Occ(\theta, A) = \{X\}$, $Occ(\theta, B) = \{X, Y\}$, $Occ(\theta,\ C) = \{Z\}$ and $Occ(\theta, P) = \emptyset$ for all other $P \in Var$. Hence, $\mathcal{A}(\theta) = \{\emptyset, \{X\}, \{X, Y\}, \{Z\}\}$.

As mentioned, the domain intuitively described above, is essentially the abstract domain of Jacobs and Langen [26]. For efficiency and increased precision, however, the analyzer integrated in the &–Prolog compiler uses the efficient abstract unification and top–down driven abstract interpretation algorithms defined by Muthukumar and Hermenegildo [36] instead of the *pure bottom–up* approach used by Jacobs and Langen.

The way in which this analyzer captures sharing information has been called *set sharing*. The power of the Sharing domain is based on this set sharing information since it not only represents when two terms possibly share, but also which variables are possibly shared and which are definitely not shared. This information allows it to accurately propagate groundness between variables. The reason is that it can represent that a set of terms share all their variables, and therefore infer the groundness of one term from the groundness of the others.

3.4 Sharing+Freeness analyzer

The Sharing+Freeness analyzer was defined in [35] as an improvement of the Sharing analyzer in which not only sharing but also freeness information was obtained. Freeness information is very useful for at least two reasons. First, the information itself is vital in the detection of *non–strict independence* [24] among goals, and also in the optimization of unification, goal ordering, avoidance of type checking, general program transformation, etc. Second, by computing this freeness information *in combination with the sharing* it is possible in turn to obtain much more accurate sharing information. Conversely, keeping accurate track of sharing also allows more precise inference of freeness. The overall effect is thus a more precise analysis than if two separate analyses were performed.

The Sharing+Freeness abstract domain approximates this information by combining two components: one is essentially the Sharing domain described above; the other encodes freeness information. The *freeness* component of an abstract substitution for a

clause gives the mapping from its program variables to an abstract domain $\{G, F, NF\}$ of freeness values [5] i.e. D_α*-freeness* $= \wp(Pvar \rightarrow \{G, F, NF\})$. X/G means that X is bound to only *ground* terms at run–time. X/F means that X is free, i.e., it is not bound to a term containing a functor. X/NF means that X is *potentially* non–free, i.e., it can be bound to terms which have functors. During the process of performing abstract unification, we use a set of temporary freeness values of the form $NF(e)$ (where e is a normalized unification equation). After abstract unification is performed, these values are changed to NF. $X/NF(e)$ means that X was free *prior* to unification by the equation $e \equiv X = f(t_1, \ldots, t_n)$ but became non–free due to the equation e. The important consequence of this is that it does not introduce any new *sharing* between the variables in $vars(f(t_1, \ldots, t_n))$ nor does it change their *freeness* values. Suppose, subsequently, that equation $e' \equiv X = Term$ (where $e \neq e'$) is processed. Now, the freeness values of X and all variables in $vars(f(t_1, \ldots, t_n))$ and $Term$ are changed from $NF(e)$ to NF. The three freeness values are related to each other by the following partial order: $\bot \sqsubseteq F \sqsubseteq NF$, $\bot \sqsubseteq G \sqsubseteq NF$

More formally, the freeness value of a term is defined as follows:

Definition 5 (Abstraction(freeness) of a Term)

$$\mathcal{A}_{freeness}(Term) = \begin{cases} \textit{if} & vars(Term) = \emptyset \quad \textit{then} \quad G \\ \textit{if} & vars(Term) = \{Y\} \wedge Term \equiv Y \quad \textit{then} \quad F \\ \textit{else} & NF \end{cases}$$

The freeness domain can be then represented as a list of those program variables which are known to be free. Its interpretation is the following:

- if a program variable X appears in the freeness component of λ, X is bound to a free variable under θ, i.e., $s.t.\ X\theta = Y, Y \in Pvar$.
- if a program variable X does not appear in the freeness component of λ, but it appears in at least one subset of the sharing component of λ, nothing can be said about the instantiation state of the term to which X is bound under θ, i.e., it can be free, ground or any complex term.

3.5 Implementation Issues

This section deals with issues that arise when implementing the analyzers described above. In particular, we will refer to the following four main issues: how the query

[5] In its most compact form, the freeness component is represented simply by the set of all program variables which are known to be free i.e. D_α*-freeness* $= \wp(Pvar)$. However, in this case, the groundness information can be obtained only from the sharing component and as a result, abstract unification algorithms become more complicated. Due to lack of space a description of the abstraction framework and unification algorithms using the $D_\alpha = \wp(\wp(Pvar)) \times \wp(Pvar)$ domain is not included here. However, it can be found in [33].

forms for the analysis are used, how available information on builtins can be taken advantage of in the process of analyzing, how meta–predicates in particular can be analyzed, and how the cut is treated in the process.

The query forms in its minimal form (least burden on the programmer) can be simply the names of the predicates which can appear in user queries (i.e., the program's "entry points"). In order to increase the precision of the analysis, query forms can also include a description of the set of abstract (or concrete) substitutions allowable for each entry point. In the actual implementation of the analyzers, the query form is a declarative–like declaration `qmode/2` which has as first argument the goal pattern of one of the entry predicates of the program and as second argument a term with functor `info` and any arity. Each of the arguments for this term corresponds to each of the analyzers of the &–Prolog system. Abstract substitutions approximating the concrete substitutions that can occur when entering the program can be placed here.

The analysis process can be viewed as interpreting the semantics of a program over an abstract domain. This is done through analyzing the definitions of the program predicates. But in doing this, predicates can be found which do not have an explicit definition in the program, i.e. builtins. Thus, the semantics of builtins must be handled by the analyzers themselves. This has the advantage of improving the information inferred by the analyzer due to the a priori knowledge that they provide, thus avoiding possible losses of information due to complex program definitions. In addition, builtins can provide control information since they are easily "abstract executable" [19] — typically, failure can be inferred at a program point where the builtin `var(X)` appears and it is known that `X` is not a variable at that point. A disadvantage of this approach is that system–specific builtins have to be added to the analyzer knowledge when porting it. But this is a quite straightforward task once the precise behavior of the builtins is known.

There is a class of builtins which are specially interesting and also difficult when analyzing a program: meta–predicates (i.e. `call/1, setof/3, findall/3, bagof/3, \+/1`). These predicates use other predicates as arguments. Thus, they often can not be manipulated because it can not be known in advance which other predicates they are calling to (i.e. those they use as arguments). In other cases the goal used as argument is made explicit in the program (typically when using `bagof/3` and the like). In these cases analysis can proceed further by taking the goal argument as the procedure call and analyzing it. Correctness of this transformation relies on the fact that the exit substitution for a procedure call correctly approximates the semantics of all possible solutions to this call, thus also approximates that of `bagof/3, findall/3` and `setof/3`. The case of `\+/1` is bit special, since the exit substitution inferred for the goal call is not the exit substitution for the `\+/1` goal, i.e. the call substitution to `\+/1` will remain the same as exit substitution. It is important to note that this does not mean that the goal call is not analyzed, and therefore it can modify the information already inferred for its program definition.

In the cases where the argument of the meta–predicate is not known, analysis cannot

proceed, therefore a warning is issued to the user. This is not only due to the fact that the call substitution for the next goal in the program is not known (the $\top$ element of the abstract domain could always be assumed), but also because the information already inferred for the definitions of the particular goals which might be called at that point may be incorrect.

Handling the cut in the analysis framework presented is much more straightforward: it is just ignored. Obviously, this does not alter the correctness of the results of the analysis. On the other hand, cut could be abstractly executed in order to increase accuracy of the analysis, by providing control information. However, it has not been already shown that the complexity inherent to make possible this enhancement would be worth. The reason for this is the difficulty present in deciding when the subgoals involved in the pruning will always succeed.

4 Parallelization Tools: Annotators

Annotators are concerned with identifying the opportunities for parallel execution in programs. The aim of the annotation process is, through different forms of dependency analysis, to partition the original program into processes that are independent (according to a particular definition of independence) and which can thus be safely run in parallel. This task is performed as source–to–source transformations of the program being annotated. Thus, these tools take Prolog programs and produce programs in which goals which can be executed in parallel are explicitly annotated and in which the sequencing relationships and safety conditions necessary to maintain the correctness and efficiency of the original program are also expressed.

Three different algorithms (MEL, UDG, and CDG) are available in the &–Prolog system for this purpose [34]. These algorithms select goals for parallel execution and, using the sufficient rules proposed for Strict IAP [21], generate the conditions under which independence is achieved and therefore independent parallel execution ensured. The result is a transformation of a given Prolog clause into an &–Prolog clause containing parallel expressions which achieve such independent and–parallelism.

The basic idea behind the algorithms is presented in the following, for a complete description see [34]. The MEL (Maximum Expression Length) algorithm creates only CGEs in its expressions to achieve parallelism, while seeking to maximize the number of goals to be run in parallel within a CGE. Goals in a clause are treated as an ordered sequence where dependency relations are identified, causing the required annotations to be performed and possibly a partition of the sequence into plain subsequences (i.e. no nested annotations are done). The UDG algorithm exploits only unconditional parallelism, i.e. only goals which can be determined to be independent at compile–time will be run in parallel. Thus, no run–time checks are generated, and therefore resultant expressions only use the & parallel operator. It represents a clause body as an Unconditional Dependency Graph (UDG).

The CDG algorithm also seeks to maximize the amount of parallelism available in a clause, without being concerned about the size of the resultant &–Prolog expression. The dependency relations among goals in a clause are treated through the use of a Conditional Dependency Graph (CDG). This algorithm uses if–then–else constructs in addition to CGEs in the resulting &–Prolog clauses. A CDG (Conditional Dependency Graph) is a directed (acyclic) graph with labelled edges. Vertices in the graph correspond to clause goals, edges to dependencies, and labels to independence conditions. An edge exists to any goal `g` from any other goal to its left in the clause so that `g` may depend on it, and the label is the condition that should hold so that this dependency would disappear. An UDG is a CDG without labels, so that dependencies will always hold (edges can be seen as having a `false` label). The UDG algorithm always treats with transitively closed graphs[6].

CDGs can be simplified in order to reduce the conditions in the labels, turn them into `false` (if the condition will never hold) or to `true` (if it will always hold — the edge is then eliminated) with the information available through different kinds of analysis an user annotations. This will be discussed in section 7.1. The simplification is also applicable to UDGs, as these are always built from CDGs, turning all conditions to `false`.

4.1 MEL Algorithm

This algorithm is based on a heuristic which tries to find out points in the body of a clause where it can be split into different expressions. One of such points is where a new variable appears. Consider a goal in a clause which has the first occurrence of a variable in that clause, and this variable is used as an argument of another goal to the right of the first one. The condition in Strict IAP that must hold for two goals which share variables establishes that these variables must be ground; obviously this is not the case for such two goals, and thus this is a point where it is not appropriate to annotate a parallel expression.

The algorithm proceeds in this manner from right to left, i.e. from the last goal of the body to the neck of the clause. The clause body is then broken into two at the points where shared new variables appear, and a parallel expression (a CGE) built for the right part of the sequence split. In proceeding backwards the underlying intention is to allow to capture the longest parallel expressions possible. An equivalent heuristic will proceed forwards but split the body at the second occurrences of new variables.

As an example, consider a clause $\texttt{h(X)} :- \texttt{p(X,Y)}, \texttt{q(X,Z)}, \texttt{r(X)}, \texttt{s(Y,Z)}$. It can be compiled (under the Strict IAP conditions for independence) into the following &–Prolog parallel expression:

[6] Although the dependency relation among goals is theoretically transitive, it is not always the case that the dependencies among goals in a clause are transitively closed. Nonetheless, no loss of parallelism is implied by considering transitively closed UDGs.

```
h(X) :- ground(X) ⇒ p(X,Y) & q(X,Z),
        indep(X,Y), indep(X,Z) ⇒ r(X) & s(Y,Z).
```

Note that the body is split at `q(X,Z)` and not at `p(X,Y)`, the largest expression being achieved in this way. Note also that the first CGE does not have the condition `indep(Y,Z)` since this condition is automatically satisfied by virtue of the fact that `Z` is a new variable.

4.2 UDG Algorithm

This algorithm starts with a graph $G(V, E)$ which is a UDG where all dependencies are unconditional. The algorithm seeks to maximize the amount of parallelism possible under these dependencies. This is achieved if for any two goals for which a dependency is not present, they are annotated to be run in parallel — thus, no loss of parallelization opportunities occurs. For this, the transitive dependency relations among goals, represented by the graph edges, are considered, and conditions upon these established. Recall in the following that the UDG is transitively closed, thus it holds that $\forall x, y \in V \cdot x \ dep^\star \ y \Leftrightarrow (y, x) \in E$.

The UDG algorithm works as follows. It first starts with the set of independent goals $I = \{p \in V \mid \forall x \in V \neg\exists(x, p) \in E\}$, those which do not have incoming edges. A set of partitions $PP = \{P \in 2^I \mid \forall p \in P \cdot \exists x \in V \cdot x \ dep^\star \ p\}$ is then built, so that there is at least one goal in $V - I$ for each of these partitions P which depends on all elements of P. These goals are grouped together so that $\forall P_i \in PP \cdot Q_i = \{x \in V \mid \forall p \in P_i \cdot x \ dep^\star \ p\}$. In this context, no loss of parallelism can occur when converting the graph into a linear (parallel) expression, if and only if $\forall P_1 P_2 \in PP$,

- $P_1 \cap P_2 = \emptyset \vee P_1 \cap P_2 = P_1 \vee P_1 \cap P_2 = P_2$
- $P_1 \cap P_2 = P_1 \Rightarrow \forall q_1 \in Q_1, q_2 \in Q_2 \cdot q_2 \ dep^\star \ q_1$

Under such conditions, the corresponding parallel expression *exp* is built up from the following rules, $\forall P_1 P_2 \in PP$ if:

- $P_1 \cap P_2 = \emptyset$ then $exp(P_1 \cup Q_1) \& exp(P_2 \cup Q_2)$
- $P_1 \cap P_2 = P_1$ then $exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$

Note that for a transitively closed graph it holds that $\forall P_1 P_2 \in PP$:

- $P_1 \cap P_2 = \emptyset \Rightarrow \forall q_1 \in Q_1, q_2 \in Q_2 \cdot \neg q_1 \ dep^\star q_2 \wedge \neg q_2 \ dep^\star q_1$
- $P_1 \cap P_2 = P_1 \Rightarrow \forall q_1 \in Q_1, q_2 \in Q_2 \cdot \neg q_1 \ dep^\star q_2$

thus, no loss of parallelism occurs using the expressions implied by the above rules.

4.3 CDG Algorithm

This algorithm is pretty close to the previous one. In this case conditional dependencies present in a CDG $G(V, E)$ are considered. To do this, all possible states of computation which can occur w.r.t. the conditions present in the graph are considered, and the body goals annotated in the best parallel expressions achievable under these conditions.

The algorithm starts with the same set I of independent goals as above. The main difference relies in that goals depending unconditionally on goals in I are not coupled to them (i.e. the close relation upon each P_i and corresponding Q_i in the UDG algorithm is not followed here). On the contrary, the CDG algorithm focuses on the conditional dependencies present in the graph.

Consider the set $D = V - I$ of dependent goals. The sets of conditions other than `false` in labels of edges between goals in I and goals in D, $IConds = \{label((p, x)) \neq \texttt{false} \mid (p, x) \in E, p \in I, x \in D\}$, and in labels of edges among goals in D, $DConds = \{label((x, y)) \neq \texttt{false} \mid (x, y) \in E, x, y \in D\}$ are built. The algorithm proceeds by incrementally building up the parallel expression exp as follows, let $I = \{p_1, \ldots, p_n\}$:

- if $D = \emptyset$ then exp is $p_1 \& \ldots \& p_n$
- if $D \neq \emptyset$, $DConds = IConds = \emptyset$ then exp is built using the UDG algorithm
- if $D \neq \emptyset$, $DConds \neq \emptyset$, $IConds = \emptyset$ then exp is $p_1 \& \ldots \& p_n, exp_1$, where exp_1 is recursively computed for $G(V - I, E_1)$ being $E_1 = E - \{(p, x) \in E \mid p \in I\}$
- if $D \neq \emptyset$, $DConds \neq \emptyset$, $IConds \neq \emptyset$ then exp is built up from the boolean combinations of the elements of $IConds$

For each boolean combination C the graph $G(V, E)$ is updated as if the conditions in C hold, that is, all conditions in labels of edges of E which are implied by elements of C are deleted and all labels with conditions which are incompatible with some element of C rewritten into `false`. Note that an edge can be removed if its label becomes void (i.e. `true`).

The parallel expressions coming out of recursively applying the algorithm after this updating are annotated as if–then–elses and combined in a simplified form.

4.4 Local Analysis

As we have mentioned before, the annotation process can be improved by using compile–time information about groundness and independence provided either by the user or by a global analyzer. However, in the absence of them the annotators are able to infer local information. The easiest way to infer this kind of information is related to the builtins, since useful groundness information can be assumed from them. For

example, after the builtin `length(X,N)` we can ensure that at execution time `N` will be bound to a ground term.

Another kind of information that can be obtained through a simple local analysis is that the variables in the body of a clause which do not appear in the head of the clause will remain free and not aliased until after their first appearance. Therefore, until this point they will be independent form any other variable.

Builtins also provide granularity information which can in turn be used to decide when a builtin should be parallelized or not. A table containing those builtins which is worth parallelizing can be built. Then the annotators would consult this table when constructing the parallel expressions.

4.5 Side–effect Analysis

In general, in an execution of &–Prolog, side–effects cannot be allowed to execute freely in parallel with other goals. A mechanism of synchronization must be provided in order to prevent a side–effect from being executed before other preceding (in the sense of the sequential operational semantics) side–effects or goals, in the cases when such adherence to the sequential order is desired, i.e. if a behaviour of the program identical to that observable on a sequential Prolog implementation is to be preserved.

In order to preserve the sequential observable behaviour, side–effects can only be executed when every subgoal to their left has been executed, i.e. when they are "leftmost" in the execution tree. However, a distinction can be made between *soft* and *hard* side–effects (a side–effect is regarded to be *hard* if it could affect subsequent execution, see [17] and [32]). This distinction allows more parallelism since a soft side–effect must be synchronized only to its left, while only hard side–effects must be synchronized both to its left and right. It is also convenient in this context to distinguish between side–effect builtins and side–effect procedures, i.e. those procedures that have side–effects in their clauses or call other side–effect procedures.

All this information is made available to the compiler, and in particular to the annotation process, by means of the analyzer described in [32]. This analyzer performs a global analysis of the program, propagating the side–effect characteristic of procedures and annotates a given procedure as:

- *pure* — meaning that it has no side effects in any of its clauses
- *soft* — meaning that it has at least one clause which has a soft side–effect builtin or a soft side–effect procedure as its subgoal, but none of its clauses contain a hard side–effect builtin or a hard side–effect procedure as its subgoal
- *hard* — meaning that it has at least one clause which has a hard side–effect builtin or a hard side–effect procedure as one of its subgoals

Furthermore, each clause of a hard side–effect procedure is annotated as either:

- *(hard,hard)* — meaning that it has a hard side–effect subgoal
- *(hard,soft)* — meaning that its only side–effect subgoals are soft
- *(hard,pure)* — meaning that it has no subgoals with side–effects

Similarly, the clauses of a soft side–effect procedure are annotated as *(soft,soft)* or as *(soft,pure)*. Needless to say, all clauses of a pure procedure are annotated as *(pure,pure)*.

5 Improving the Global Analysis Tools

This section is aimed at describing the improvements performed by us to the global analysis tools developed for the &-Prolog System. An important component of our work in the area has mainly been performed in the context of a different ESPRIT project ("PRINCE"). Thus, it will not be explicitly part of the deliverables. However, given the relevance to our work in the current project, in this report we will briefly summarize such work and provide appropriate references.

5.1 ASub analyzer

The domain ASub presented in [41] was aimed at inferring more accurate sharing by adding linearity information. A program variable is *linear* if it is bound to a term which contains only single occurrences of variables.

The ASub domain approximates this information by combining two components (G, R): one representing groundness information; the other representing sharing and linear information. The concretization function, $\gamma_{ASub} : \mathsf{ASub} \rightarrow 2^{Sub}$, is defined by

$$\gamma_{ASub}(G,R) = \left\{ \theta \;\middle|\; \begin{array}{l} \forall (x,y) \in \mathsf{PVar}^2 : \\ \quad (\, x \in G \Rightarrow ground(x\theta)\,) \quad \wedge \\ \quad (\, x \neq y \wedge vars(x\theta) \cap vars(y\theta) \neq \emptyset \Rightarrow \\ \qquad x \; R \; y\,) \quad \wedge \\ \quad (\, x \not R \; x \Rightarrow linear(x\theta)\,) \end{array} \right\}$$

In other words, if a program variable X appears in G, X is known to be bound to a ground term. If (X, X) appears in R, X is possibly bound to a non linear term. If (X, Y) appears in R ($Y \in PVar$), the terms to which X and Y are bound to may share.

The advantage of the ASub domain is that it captures information about linearity which is not captured in Sharing. In ASub whenever a term is known to be linear it is possible to infer that the variables contained in the term do not share, while in

Sharing (and thus in Sharing+Freeness such sharing must be assumed. On the other hand, the Sharing domain is more powerful in the way groundness is propagated between variables. The reason is that ASub only represents when two terms possibly share, i.e. it does not provide any information about covering, which Sharing does. For this reason, the way in which ASub encodes sharing has been called *pair sharing* in contrast to the set sharing encoded by the Sharing domain.

As what happened with the Sharing+Freeness algorithm, for efficiency and increased precision, the ASub analyzer integrated in the &–Prolog compiler uses the efficient abstract unification by Codish et al [6] instead of the approach used by Søndergaard.

5.2 Combining Domains

As we have seen in previous sections, it is often the case that program analyses aim to provide a combination of more basic types of information. Typically, such combined analyses provide more information than that obtained by combining the information which might be obtained by the corresponding individual analyses. Moreover, efficiency can also improve as the increased precision reduces the number of irrelevant analysis paths which the abstract computation is obliged to follow. However, the design, implementation and formal justification of such combined analyses require new efforts which do not directly benefit from previously designed analyses.

In [7] we observe that in many cases it is possible to provide combined analyses which do benefit from previously defined analyses and which also maintain a high degree of precision. The theoretical background for the work has been laid down already in 1979 by Cousot and Cousot [11]. There, the authors illustrate that although some precision can be gained by removing redundancies from combined domains, still further precision is gained by introducing new basic operations. The subject of the work presented in [7] is centered around the practicality of: Given an appropriate concrete semantics and having found two or more notions of description together with corresponding approximations of the basic operations in the concrete semantics, *automatically* constructing an approximation of the basic operations for a combined notion of description. Given this construction a combined analysis is derived by abstracting the concrete semantics.

Direct product analysis
Let E be a concrete domain and let $(E, \alpha_i, D_i, \gamma_i)$ $i \in \left\{ 1, 2 \right\}$ be Galois insertions. The *direct product domain* is a quadruple $(E, \alpha_\times, D, \gamma_\times)$ where $D = D_1 \times D_2$, $\gamma_\times : D \to E$ is defined by $\lambda(d_1, d_2).\ \gamma_1(d_1) \sqcap_E \gamma_2(d_2)$ and $\alpha_\times : E \to D$ is defined by $\lambda e.\ (\alpha_1(e), \alpha_2(e))$.

The direct product domain is not a Galois insertion. Intuitively, the problem comes from the existence of redundant information in the abstract domain. On the other hand, given a function $\mu : E \to E$ and corresponding γ_i-approximations $\mu_i^{\mathcal{A}} : D_i \to D_i$ for $i \in \left\{ 1, 2 \right\}$, the *direct product function* $\mu_\times^{\mathcal{A}} : D \to D$ defined by $\lambda(d_1, d_2).\ (\mu_1^{\mathcal{A}}(d_1), \mu_2^{\mathcal{A}}(d_2))$ is a $\gamma_\times$-approximation of μ.

Intuitively, the direct product function corresponds to performing the analyses $\mu_1^{\mathcal{A}}$ and $\mu_2^{\mathcal{A}}$ independently, and then "group" the information. It is clear that this combined analysis potentially provides more information than obtained by each analyzer. However, it does not improve the information obtained by each analyzer itself.

Reduced product analysis
Let $(E, \alpha_i, D_i, \gamma_i)$, $i \in \left\{ 1, 2 \right\}$ be Galois insertions and let $(E, \alpha_\times, D, \gamma_\times)$ be the corresponding direct product domain. The relation $\equiv \ \subseteq D \times D$ induced by $\gamma_\times$ is defined by $d \equiv d' \Leftrightarrow \gamma_\times(d) = \gamma_\times(d')$.

Intuitively, this relation induces equivalence classes in D, each class containing those elements which are redundant, i.e. which approximates the same set of concrete substitutions.

The *reduced product domain* is a quadruple $(E, \alpha_\star, D_\equiv, \gamma_\star)$ where $\alpha_\star : E \rightarrow D_\equiv$ is defined by $\lambda e.\ [\alpha_\times(e)]_\equiv$ and $\gamma_\star : D_\equiv \rightarrow E$ is defined by $\lambda[d].\ \gamma_\times(d)$. It is straightforward to show that the reduced product domain is well defined and is a Galois insertion.

Let $\mu : E \rightarrow E$ be a concrete function and let $\mu_i^{\mathcal{A}} : D_i \rightarrow D_i$, $i \in \left\{ 1, 2 \right\}$, be corresponding γ_i-approximations. The *reduced product function* $\mu_\star^{\mathcal{A}} : D_\equiv \rightarrow D_\equiv$ is defined by

$$\lambda[d]_\equiv.\ [(\mu_1^{\mathcal{A}}(d_1), \mu_2^{\mathcal{A}}(d_2))]_\equiv$$

where $(d_1, d_2) = \sqcap_D [d]_\equiv$, namely the smallest representative of the equivalence class $[d]_\equiv$.

In other words, the reduced product corresponds to perform the analyses independently with one special characteristic: any time that the reduced product function over an element $d \in D$ has to be applied, the smallest reepresentativee of the equivalence class $[d]_\equiv$ will be chosen and the reduced product function will be applied to it. Thus redundant information will be avoided. It is straightforward to show that the reduced product function $\mu_\star^{\mathcal{A}}$ is well defined and is a $\gamma_\star$-approximation of μ. Moreover, in general the reduced product function is more precise than the direct product function. It is important to note that, since no abstract function has been modified, both the implementation and the formal justification of each individual analysis can be reused.

As a simple example for logic programs, [7] illustrats how the ASub domain can be represented as the reduced product of corresponding sharing and groundness domains. The resulting reduced product analysis is equivalent to that derived from the abstract unification of Codish et al. [6] for this domain.

This approach has been implemented in the &–Prolog compiler by combining the Sharing and ASub domains and the Sharing+Freeness and ASub domains. There are strong indications that this automatically combined analyses in fact compare well with other new proposals suggested in recent literature [9, 42] both from the point of view of efficiency and accuracy.

6 Improving the Parallelization Tools

The annotation algorithms described in section 4 can be generalised so that they become isolated from the concept of independence under consideration. This allows integrating them in a parallelization framework where different notions of independence are used. From this point of view, the annotation process can be seen as composed of two distinct sub–processes: a front–end which first establishes the conditions for independence of goals and makes dependencies explicit in a suitable structure, and a back–end which then manages this structure to come up with an &–Prolog expression. The first sub–process will be an independence–checker, while the second will be an expression–annotator, or simply a paralleliser. The algorithms presented in section 4 are reviewed here from this perspective as expression–annotators, once independence has been computed using a separate process. Also a new algorithm of this style is presented for Non-Strict IAP, which, combined with a new independence–checker to be implemented using the theory presented in section 6.6 based on the Sharing+Freeness analyser, will provide a complete compile–time paralleliser capable of exploiting Non-Strict IAP.

The generalised MEL algorithm is presented below. The CDG algorithm is based on a *update* function which rewrites conditions in the labels of the graph based on conditions that hold and a boolean *combine* function which computes combinations of these conditions (see section 4.3). Except for these two functions, the algorithm is in itself general enough to be applied to any notion of independence. The UDG algorithm is also general in this sense.

On the other hand, the UDG algorithm is focussed on perfect linearization of a graph, being the main objective not to lose oportunities for parallelism (see section 4.2). Unfortunately, this can not always be achieved. The conditions under which it can be achieved do not always hold in real programs, thus the algorithm had to be extended to cope with other situations.

Also, for the CDG algorithm some variants of it can be taken into account. The basic annotation heuristic for goals known to be independent can be re–defined in different terms. On one hand, it can be pushed closer to the UDG algorithm, on the other hand, it can be easily replaced with other heuristics.

6.1 Generalising MEL

The MEL algorithm is presented in this section in such a way that its main heuristic has been made independent of Strict IAP. The algorithm is then applicable with wider generality.

Let $C = H :- B$ and $B = B_1, \ldots, B_n$. Define $\mathcal{K} = \{\mathcal{K}(B_i)\}$ where $\mathcal{K}(B_i)$ is the set of the relevant items of information for the independence notion under consideration that are known to be true *before* executing B_i. As explained in sections 4.4 and 7.2, $\mathcal{K}$ can

be computed from a combination of global analysis based on abstract interpretation, local analysis of the clauses bodies and user–supplied information.

Define $\mathcal{I} = \{\mathcal{I}(B_i, B_j) \mid i \leq j\}$, where $\mathcal{I}(B_i, B_j)$ are the sets of conditions such that B_i and B_j are independent, and these conditions could hold at execution–time in view of the known facts in $\mathcal{K}(B_i)$.

The algorithm starts with a sequence B of goals (initially the body of the clause under consideration) and computes its correspondent parallel expression D as follows:

1. Let $B = B_1, \ldots, B_q$. Find the largest p such that for some $B_i, p < i \leq q$: $\mathcal{I}(B_p, B_i) = \{false\}$.
   If there is no such p, then let $p = 0$, $B1 = null$ and $B2 = B$. Else, let $B1 = B_1, \ldots, B_p$ and $B2 = B_{p+1}, \ldots, B_q$.
2. Let $IConds = \bigcup_{q>i>p\ q\geq j>i} \mathcal{I}(B_i, B_j) - \mathcal{K}(B_{p+1})$ be the set of conditions for goals in $B2$ to be independent, except those that are known just after the execution of B_p.
3. Therefore, the &–Prolog parallel expression corresponding to $B2$ is
   $$D2 = ((\bigwedge_{x \in IConds} goal(x)) \Rightarrow B_{p+1} \& \ldots \& B_q)$$
   where $goal(x)$ is the &–Prolog goal that satisfies the condition x.
4. If $B1 = null$, then $D = D2$. Else, $D = D1, D2$ where $D1$ is the &–Prolog parallel expression corresponding to $B1$.

As an example, consider the clause $\mathtt{C} = \mathtt{h(X)} :- \mathtt{p(X,Y)}, \mathtt{q(X,Z)}, \mathtt{r(X)}, \mathtt{s(Y,Z)}$ (the same as in section 4.1). Under the Strict IAP notions of independence, we have the following:

$\mathcal{K}(\mathtt{p(X,Y)}) = \{\mathtt{free\_not\_aliased(Z)}\}$
$\mathcal{I}(\mathtt{p(X,Y)}, \mathtt{q(X,Z)}) = \{\mathtt{ground(X)}, \mathtt{indep(Y,Z)}\}$
$\mathcal{K}(\mathtt{q(X,Z)}) = \{\mathtt{free\_not\_aliased(Z)}\}$
$\mathcal{I}(\mathtt{q(X,Z)}, \mathtt{r(X)}) = \{\mathtt{ground(X)}\}$
$\mathcal{I}'(\mathtt{q(X,Z)}, \mathtt{s(Y,Z)}) = \{\mathtt{ground(Z)}, \mathtt{indep(X,Y)}\}$
$\mathcal{I}(\mathtt{q(X,Z)}, \mathtt{s(Y,Z)}) = \{\mathtt{false}\}$
$\mathcal{K}(\mathtt{r(X)}) = \emptyset$
$\mathcal{I}(\mathtt{r(X)}, \mathtt{s(Y,Z)}) = \{\mathtt{indep(X,Y)}, \mathtt{indep(X,Z)}\}$

So C can be compiled into the following &–Prolog parallel expression:

```
h(X) :- ground(X) ⇒ p(X,Y) & q(X,Z),
        indep(X,Y), indep(X,Z) ⇒ r(X) & s(Y,Z).
```

Note that the first CGE does not have the condition `indep(Y,Z)` since this condition is automatically satisfied by virtue of the fact that $\mathtt{free\_not\_aliased(Z)} \in \mathcal{K}(\mathtt{p(X,Y)})$.

6.2 Extending UDG

Recall from section 4.2 that the UDG algorithm is based on a set PP of partitions of the set of actual independent goals at one moment in the iteration of the algorithm. From the definition of this partitions, note that if always holds that $\forall P_1 P_2 \in PP \cdot P_1 \neq P_2$ and either:

- $P_1 \cap P_2 = \emptyset$
- $P_1 \cap P_2 = P_1$
- $P_1 \cap P_2 = P \mid P \neq \emptyset, P \neq P_1, P \neq P_2$

The UDG algorithm finds out the best linearization of the dependency graph in such a way that no loss of parallelism occurs. For this to be possible, the body of a given clause must fit into either the first case or a special sub–case of the second one. In order to extend the algorithm to deal with all possible cases, the following possible graph linearizations have to be considered:

1. $P_1 \cap P_2 = \emptyset$

   $$exp(P_1 \cup Q_1) \& exp(P_2 \cup Q_2)$$

2. $P_1 \cap P_2 = P_1$

   (a) $\forall q_1 \in Q_1, q_2 \in Q_2 \cdot q_2 \ dep^\star \ q_1$

   $$exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$$

   (b) $\forall q_1 \in Q_1, q_2 \in Q_2 \cdot \neg q_2 \ dep^\star \ q_1$

   i. at the loss of parallelism between Q_1 and Q_2

   $$exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$$

   ii. at the loss of parallelism between Q_1 and $P_2 - P_1$

   $$exp(P_1) \& exp(P_2 - P_1), exp(Q_1) \& exp(Q_2)$$

   (c) $\exists q_1 \in Q_1, q_2 \in Q_2 \cdot q_2 \ dep^\star \ q_1$

   i. at the loss of parallelism between $q_2 \in Q_2$ and $q_1 \in Q_1$ s.t. $\neg q_2 \ dep^\star \ q_1$

   $$exp(P_1 \cup Q_1) \& exp(P_2 - P_1), exp(Q_2)$$

   ii. at the loss of parallelism between Q_1 and $P_2 - P_1$

   $$exp(P_1) \& exp(P_2 - P_1), exp(Q_1 \cup Q_2)$$

iii. being $Q_{12} = \{q_1 \in Q_1 \mid \exists q_2 \in Q_2 \cdot q_2\ dep^\star\ q_1\}$ and $Q_{11} = Q_1 - Q_{12}$

$$exp(P_1 \cup Q_{12}) \& exp(P_2 - P_1), exp(Q_{11}) \& exp(Q_2)$$

at the loss of parallelism between Q_{11} and $P_2 - P_1$ and also between $q_2 \in Q_2$ s.t. $\forall q_1 \in Q_1 \cdot \neg q_2\ dep^\star\ q_1$ and Q_{12}

3. $P_1 \cap P_2 = P \mid P \neq \emptyset, P \neq P_1, P \neq P_2$

$$exp(P_1 \cup P_2), exp(Q_1 \cup Q_2)$$

at the loss of parallelism between $q_2 \in Q_2$ and $p_1 \in P_1 - P$ and also $q_1 \in Q_1$ and $p_2 \in P_2 - P$

The original UDG algorithm deals with cases 1 and 2a. The natural extension of the algorithm to be able to deal with the whole of case 2 is to make it force the assumption that the required condition in case 2a holds and let it behave as in this case. This leads the extended algorithm to take into account cases 2(b)i and 2(c)i. To make the extension complete, it has to also deal with case 3, for which the partitions involved are considered as a single one: a new partition is built up from the union of the other ones. Note that each of these extensions implies a loss of parallelism.

6.3 Improvements to UDG

The proposed extension to the UDG algorithm seems to be its natural extension when the objective of maximizing possibilities for parallelism is left aside. Nonetheless, when it is considered w.r.t. the execution efficiency of the parallel expressions it obtains, it turns out that the extension must be made in another direction.

This can be seen with a simple experiment. Consider all the possible parallel expressions listed above for a situation like case 2 and let $P_1 = \{p_1\}$, $P_2 = \{p_1, p_2\}$, $Q_1 = \{q_1\}$ and $Q_2 = \{q_2\}$ be the minimum sets that fulfill the conditions in that situation. Now, consider giving to each of these goals a measure of an upper bound of its granularity and computing all possible combinations of granularities of all the goals. The efficiency of each of the possible parallel expressions proposed above can be computed and are shown in table 1. There the percentage of combinations where each parallel expression behaves best is shown. The percentage includes the situations where all the expressions behave the same, thus the total percentage can sum up more than 100%.

From the table, it is evident that the best parallelization strategy corresponds to the second option in both cases. This is due to the fact that this strategy performs a better load balancing of parallel tasks with goals which are already balanced (i.e. have almost the same granularity, as with maximum grain of 1 or 2) or for which the differences in grain are not high. When a bigger difference is allowed (increasing the maximum permitted goal grain) the mean efficiency of this strategy lowers a bit, while the first

Max.G. Grain	% Best case	
	2(b)i	2(b)ii
1	0	100
2	18	100
3	22	97
4	23	95
5	24	94
6	24	93
7	24	92
8	24	92
9	24	91
10	24	91

Max.G. Grain	% Best case		
	2(c)i	2(c)ii	2(c)iii
1	0	100	0
2	12	100	12
3	15	96	12
4	16	93	11
5	16	92	11
6	17	91	10
7	17	90	10
8	17	89	10
9	17	88	10
10	17	88	9

Table 1: Performance test for possible parallel expressions

strategy (that pointed out in the previous section) progressively behaves better, but in any case the asymptotic values seem to stabilize.

Therefore, the best parallelization strategy, which should be used to extend the UDG algorithm, should be that of cases 2(b)ii and 2(c)ii. In both cases the strategy is the same: a parallel expression is built up for the P_i goal partitions and separately for the Q_i ones and these two are annotated for sequential execution.

It is worth noting that this result points out the importance of having granularity information on the goals being annotated, so that the annotators could take granularity into consideration in the load balancing algorithms. Unfortunately, having good measures for the granularity of goals is a difficult task. In the absence of information on granularity, the parallelization strategy herein presented should be pursued.

On the other hand, it turns out that an algorithm exploiting this strategy has to locally consider the goals to be annotated. When the whole of a clause is considered (as it is the case in an UDG), many different instances of the possible cases listed above arise. If a grouping of goals such as the partitions the UDG algorithm makes is done, the strategy does not perform as intended. Note that the cases in which this strategy is based suggest parallel expressions that will be obtained with the UDG original strategy if all elements of $Q_1 \cup Q_2$ depend on all elements of $P_1 \cup P_2$. Thus, the new strategy suggests a grouping of partitions which when applied to a whole clause derives in too many "imaginary" dependencies being created. Therefore, this strategy should be applied in a goal–to–goal fashion, incrementally creating the required dependencies based on a pairwise consideration of goals. An approach to an algorithm in this style is presented below in section 6.5.

6.4 Improvements to CDG

The CDG algorithm is focussed on annotating all possible parallel expressions that can be exploited based on the different situations that may occur at execution–time. In doing this, it avoids dealing with unconditional dependencies and focusses on conditions which can allow independence of goals. Thus, recall from section 4.3 that when it is the case that $D \neq \emptyset$, $DConds \neq \emptyset$ and $IConds = \emptyset$ then an unconditional parallel expression was built for elements in I followed sequentially by another expression recursively computed for the rest of the goals. No consideration is done in this case about the unconditional dependencies that could occur from other goals to goals in I.

The UDG algorithm, on the other hand, does this and groups goals depending unconditionally on those of I together and with those on which they depend, building an expression for the different groups of goals made. An extension of the CDG algorithm in this direction will extract from the CDG the subgraph of unconditional dependencies on goals of I (for the case mentioned above) and behave as the UDG algorithm, then returning back with an updated graph to the original algorithm. This extension will allow a one–to–one correspondence between both algorithms, so that expressions built up by each of them will be the same, modulo the conditions present in conditional expressions.

Further extensions of the CDG algorithm in the same direction can be done. The heuristic that this algorithm exploits of considering all combinations of conditions can be replaced by the UDG algorithm strategy of partitioning the graph into strongly dependent groups of goals. In this case the effect would be that the parallel expressions being annotated will look like conditional expressions nested inside unconditional ones. This turns out to be a different algorithm than that of the CDG: it will consist in the same UDG algorithm performing not only on unconditional edges of the graph but also on conditional ones, but in turn annotating these latter with the required conditions.

A different issue arises at the aim of allowing the annotating algorithms to be used for different notions of independence. In this case, it has to be noted that both the CDG and UDG algorithms freely alter the original order of the goals in a clause when they are independent. This is always done under the assumption that being the goals independent, they can be run in parallel, and thus its order in the program text is not relevant. Although this can be true for Strict IAP, since it is a transitive notion, it is not applicable for Non-Strict IAP, where the conditions depend on the order of the goals involved (see [23]).

Altering the order of goals in parallel expressions can also be inefficient in backwards execution in &–Prolog (it can cause undesired recomputation of goals when backtracking into the parallel expression — as pointed out by [13]). For example, consider two goals `p, q` in a clause body such that `p` has failure derivations and one answer and `q` has n answers with $n \geq 2$. If it is annotated as `p & q` and failure occurs after executing this expression, the failing branch of `p` will only be executed once, since the backtracking

strategy of &–Prolog in this case resembles that of standard Prolog. If, on the other hand, `q & p` is executed, the failing branch of `p` will be computed once for any answer of `q`.

6.5 A new approach to annotation for Non-Strict IAP: the URLP algorithm

Because of the problem of the reordering of goals in CDG/UDG algorithms, for Non-Strict independence either the modification of these algorithms or a new annotator was needed. The second approach was taken because it seemed more difficult to completely adapt the mentioned algorithms to avoid such reordering than to develop a new algorithm, and because a simpler algorithm is desirable since it can be easily adapted to incorporate new information from the analyses.

As the first implementation of Non-Strict IAP will be unconditional (no runtime checks), the new algorithm of annotation was firstly made accordingly — but we think that the algorithm, due to its simplicity, can be easily adapted to deal with such checks. The algorithm is heuristic, but showed similar degree of parallelization as UDG, perhaps because the UDG algorithm is heuristic except when the conditions mentioned in section 4.2 are met.

Also, because of the requirement of preserving the original order of the goals, the interface with the independence–checker is different, since it is not possible receiving only a dependency graph. The new algorithm, named URLP (Unconditional Recursive Linear Parallelizer) starts with the sequence of goals of the body of the clause and recursively tries to parallelize pairs of consecutive goals or groups of goals by a few simple rewriting rules, based on the dependencies found by the independence–checker. URLP proceeds applying from left to right the following rewriting rules, until no change can be done:

Rule	Pattern	Condition	New Pattern
1	...A, B...	¬ dep(A,B)	...A & B...
2	...PA, B...	(1)	...IA & (DA, B)...
3	...PA, PB...	(2)	...(PA, DB) & IB...

Where A and B represent single goals and PA, PB, DA, DB, IA and IB represent single goals or parallel expressions.

(1) IA (DA) is the parallel expression formed with the elements of PA from which B is independent (dependent). After the rule is applied, "DA, B" is parallelized recursively.

(2) IB (DB) is the parallel expression formed with the elements of PB which are independent (dependent) from PA.

Since at each step a simple rule is applied, it can be easily shown that the parallelization is correct. To see if it is optimal, however, we would need the execution times of

the goals in the general case.

As an example, consider the parallelization of the body clause "a, b, c, d, e, f", where the dependencies found by the annotation front–end (the independence–checker) are dep(a, d), dep(a, e), dep(b, d), dep(c, f) and dep(e, f). The steps followed by the algorithm are shown below:

Step	Expression	Apply Rule #
1	a, b, c, d, e, f	1
2	$a\&b, c, d, e, f$	2
3	$a\&b\&c, d, e, f$	2
4	$(a\&b, d)\&c, e, f$	2
5	$(a\&b, d, e)\&c, f$	1
6	$(a\&b, d\&e)\&c, f$	

The parallelization provided by UDG is $b\&c\&(a, e), d\&f$, reversing the relative order of execution of the goals e and d, which is what must be avoided for Non-Strict independence.

6.6 Towards Extracting Non-Strict IAP Using Sharing+Freeness Information

Despite the advantage of Non–Strict Independence (NSI) over Strict Independence (SI) in terms of generality and the amount of parallelism it can exploit, all compilation technology developed to date has been based on SI, presumably due to the complexity of compile–time detection of NSI.

In [5], we have developed a technique for compile–time detection and annotation of Non-Strict IAP, based on the Sharing+Freeness analysis. The technique covers the sufficient conditions found for compile–time detection of Non-Strict Independence, along with algorithms for combined compile–time/run–time detection, presenting novel run-time checks for this type of parallelism. Another important issue resolved is an efficient algorithm for performing combined compile–time/run–time renaming of shared variables among parallel goals, which is needed for the parallel execution of non–strictly independent goals [23]. Finaly, new approaches of abstract domains are proposed which we believe would improve the accuracy of the conditions.

With the implementation of these algorithms and the integration of them into the &–Prolog compile–time system, we will have a complete paralleliser capable of exploiting Non-Strict IAP.

7 Integrated Compile–time System

In this section we have an overview of the current state in the implementation of the compile–time system of &-Prolog. All tools reviewed in this report are being integrated

in a unified environment for parallel program development, having as test–bed the &-Prolog run–time system, and as interface a common language. Integration is being carried out in a modular framework, which allows for ease of future incorporations of tools to the system, as well as for comfortable testing of such tools.

The integrated system is able to take an &-Prolog program and perform selected analysis and annotation of it. Currently, the whole compilation process can be performed with only local analysis information, with global analysis information, or both, plus additional information the user can supply. The aim in this is to provide a uniform framework for supplying the required information to the process. This information can be automatically inferred, but can also be supplied by the programmer. The same happens with the language: annotations can be done explicitely by the programmer, or left to the automatic tools.

The compilation process ends with incore (executable) code. Nonetheless, it is important to note that the annotated code and the compiled byte code can be saved into a file, in addition or as an alternative to being executed in parallel on the host multiprocessor. This serves several purposes: the code can be used as the input for a series of high– and low–level simulators. Also, the user can thus see the output of the parallelization stages and monitor the compiler if desired.

7.1 Information for the Annotation Process

The usefulness of compile–time information for various program optimizations is generally agreed by compiler implementors. The annotation process is not an exception. Information on the instantiation degree of variables, types of terms, sharing between variables and the like can help annotating the program in many ways.

First, all available information of that kind is crucial in accurately identifying data–dependencies between goals. In the absence of such information, data–dependencies have to be assumed from every goal in a clause to every other goal to its right. The conditions for these dependencies to disappear are subject to the independence upon which the process is operating. Having information on the run–time states of the program can help in simplifying these conditions. The simplification can be done by:

- identifying and eliminating tests that will always succeed, and thus allowing the annotators to exploit unconditional parallelism,
- identifying tests that will never succeed and marking them by unavoidable dependencies, allowing the annotators to "pass over" such strongly dependent goals, optimizing the annotated expressions,
- identyfing tests that are redundant, by propagating the conditions already placed in the expressions w.r.t. the information available, and shortening the checks, allowing the annotators to build more efficient expressions.

Only the first simplification of the above was clearly identified and taken advantage of it in the annotators. Currently, they are prepared to exploit all of them, by means of a pre-process, which, in addition to performing local analysis, gathers all available information and simplifies the data–dependencies in the ways mentioned.

7.2 Global and Local Analysis and User Information

In the current implementation two modules are responsible for gathering all possible information about the program. A first module performs global analysis based on an abstract interpretation framework (defined in section 3, which is itself modular in its own right. A second module, interfaced with the former, is responsible for gathering global analysis information and user–supplied information and performing local analysis. All the information is then passed over to a third module which identifies and simplifies data–dependencies — the independence checker mentioned in section 4. This third module is being parameterized in the concept of independence which is to be exploited, which will allow us to plug in different checkers for the different concepts we are currently investigating.

The global analysis module has been improved in many different directions, as explained in section 5. Not only have the domains extended, and the framework isolated (allowing incorporation of many other different domains) and enhanced (for better accuracy and efficiency), but also the combination of different domains have been made possible within the same framework. In the same direction, the local analysis module is being isolated from subsequent phases of the annotation process, which will allow for more efficiently performing this task and integrating all possible means of analysing the program. The interface for information–gathering phases has been defined and tools are to be adapted to this interface in the very next future. This same interface allows for user–supplied information to be expressed in the program source. The interface is defined via an abstract syntax, which is discussed in the following.

7.3 Interface Analysis/Annotation: New Abstract Syntax

The interfaces between the different compilation tools in the system has been defined in terms on an abstract syntax which will help the manipulation of programs at compile–time, as well as the exchange of sources and tools for this manipulation, be them the different &-Prolog tools or other external tools that may conform this syntax. For this purpose, a pre–process module has been added to the compiler in order to conform to the syntax, in addition to adapt all the tools to the defined interface, which is being currently carried out.

The abstract syntax is defined in *ParForce* deliverable D.WP1.1.M1. There, a very simple Edinburgh Prolog syntax is defined, to take the tools to an agreement for exchanging of sources, and a wealth of declarations are proposed, which define the interface for these tools. Such declarations serve the two main purposes for which they have

been conceived: to give means to (maybe very) different compile–time tools to communicate knowledge about the program between them, and to allow the user to facilitate the task of these tools by providing them with additional information. Nonetheless, most declarations of properties are optional for the programmer. But in certain cases some compilation tools, as global analysis based on abstract interpretation, can not proceed with the required accuracy. Thus, in these cases the user should be warned that no refinement of his/her program can be done unless he/she supplies the required information. These issues are also discussed in the mentioned report.

References

1. K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, October 1990.

2. P. Biswas, S. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR/Restricted AND Parallelism in Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, Seattle,Washington, 1988.

3. P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

4. M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

5. D. Cabeza and M. Hermenegildo. Towards Extracting Non-strict Independent Andparallelism Using Sharing and Freeness Information. Technical Report TR Number CLIP5/92.1, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 1993.

6. M. Codish, D. Dams, and E. Yardeni. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In *Eighth International Conference on Logic Programming*, pages 79–96, Paris, France, June 1991. MIT Press.

7. M. Codish, A. Mulkers, M. Bruynooghe, M.J.García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

8. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

9. A. Cortesi and G. File. Comparison and Improving: Abstract Domains for Sharing Analysis. Technical report, Dipartimento di Mathematica pura ed applicata, Universitá degli studi di Padova.

10. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Acm Symp. on Principles of Programming Languages*, pages 238–252, 1977.

11. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

12. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

13. M.J.García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium.* MIT Press, Boston, MA, October 1993.

14. S. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. Technical Report 87-24, Dept. of Computer Science, University of Arizona, August 1987.

15. S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

16. D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

17. D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

18. S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.

19. F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Parallel Language Implementation and Logic Programming*, pages 323–335. Springer-Verlag, 1991.

20. G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

21. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

22. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. Technical Report ACA-ST-032-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1989.

23. M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

24. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1993. To appear.

25. M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

26. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

27. N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

28. A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.

29. Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

30. H. Mannila and E. Ukkonen. Flow Analaysis of Prolog Programs. In *Fourth IEEE Symposium on Logic Programming*, pages 205–214, San Francisco, California, September 1987. IEEE Computer Society.

31. C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.

32. K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

33. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. Technical Report STP-368-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, December 1990.

34. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

35. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

36. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

37. F. Nielson. Strictness analysis and denotational abstract interpretation. *Information and Computation*, 76(1):29–92, 1988.

38. B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

39. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

40. K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, October 1991.

41. H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

42. R. Sundarajan and J.S Conery. An Abstract Interpretation Scheme for Groundness, Freeness, and Sharing Analysis of Logic Programs. In *12th FST & TCS Conference*, New Delhi, India, December 1992.

43. P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

44. A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 700–710, Seattle,Washington, August 1988.

45. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

46. D. H. D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

47. D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.

48. R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Washington, August 1988. MIT Press.

49. H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.